Giving MoarVM a new general dispatch mechanism to speed up various slow Raku constructs

Jonathan Worthington
Edument
What is dispatch and why does it matter?

How we approached dispatch thus far and the shortcomings of past approaches

A new generalized approach to implementing the many different kinds of dispatch

The current status in terms of completion and performance

Future opportunities for further improvements
What is dispatch and why does it matter?
$store.get-product($id)

```java
class Store {
    method get-product($id) {
        ...
    }
}
```
$x + y$

multi infix:«+»(Int $x$, Int $y$) {
    ...
}

?
Dispatch fills the places in-between the code we write
It's everywhere...

...but we'd like to see it nowhere

(especially not high on profiler output)
More generally, dispatch is any process where the types or values of arguments determine what code we run.
Assignment is dispatch

# Just write to the Scalar $!value attribute
my $x = $v;
# Depends on the type of $v (write or error)
my Int $y = $v;
# Reset to the default value
$x = Nil;
# May need to vivify the hash value
%h<x> = $v;
An incomplete list of things that are essentially dispatch in Raku

- Standard method calls ($o.m)
- Maybe method calls ($o.?m)
- Qualified method calls ($o.T::m)
- Private method calls ($o!pm)
- Qualified private method calls ($o!T::Pm)
- Multiple dispatch
- Invocation of an object (Code, CALL-ME)
- callsame, nextwith, etc.
- Anything that has been wrap'd
- Coercion
- Return type assertion
- Binding type assertion
- Assignment
- Sinking
And many of these combine
(for example, a wrapped multi method)

Standard method calls ($o.m)
Maybe method calls ($o.?m)
Qualified method calls ($o.T::m)
Private method calls ($o!pm)
Qualified private method calls ($o!T::Pm)
Multiple dispatch
Invocation of an object (Code, CALL-ME)
callsame, nextwith, etc.
Anything that has been wrap'd
Coercion
Return type assertion
Binding type assertion
Assignment
Sinking
How we approached dispatch thus far and the shortcomings of past approaches
In the beginning...
In the beginning...

```
DISPATCH(NEXT_OP) {
  OP(no_op):
    goto NEXT;
  OP(const_i64):
    GET_REG(cur_op, 0).i64 = MVM_BC_get_I64(cur_op, 2);
    cur_op += 10;
    goto NEXT;
  OP(add_i):
    GET_REG(cur_op, 0).i64 = GET_REG(cur_op, 2).i64 + GET_REG(cur_op, 4).i64;
    cur_op += 6;
    goto NEXT;
  OP(if_i):
    if (GET_REG(cur_op, 0).i64)
      cur_op = bytecode_start + GET_UI32(cur_op, 2);
    else
      cur_op += 6;
    GC_SYNC_POINT(tc);
    goto NEXT;
  ...
```
In the beginning...

```
DISPATCH(NEXT_OP) {
    OP(no_op):
        goto NEXT;
    OP(const_i64):
        GET_REG(cur_op, 0).i64 = MVM_BC_get_I64(cur_op, 2);
        cur_op += 10;
        goto NEXT;
    OP(add_i):
        GET_REG(cur_op, 0).i64 = GET_REG(cur_op, 2).i64 + GET_REG(cur_op, 4).i64;
        cur_op += 6;
        goto NEXT;
    OP(if_i):
        if (GET_REG(cur_op, 0).i64)
            cur_op = bytecode_start + GET_UI32(cur_op, 2);
        else
            cur_op += 6;
        GC_SYNC_POINT(tc);
        goto NEXT;
    ...}
```
Interpreting bytecode is rather slow...

...but C is pretty darn fast...

...so write the performance critical parts in C
Thus, complex ops...

OP(findmeth): {
    /* Increment PC first, as we may make a method call. */
    MVMRegister *res = &GET_REG(cur_op, 0);
    MVMObject *obj = GET_REG(cur_op, 2).o;
    MVMString *name = MVM_cu_string(tc, cu, GET_UI32(cur_op, 4));
    cur_op += 8;
    MVM_6model_find_method(tc, obj, name, res, 1);
    goto NEXT;
}

Thus, complex ops...

OP(findmeth): {
/* Increment PC first, as we may make a method call. */
MVMRegister *res = &GET_REG(cur_op, 0);
MVMObject *obj = GET_REG(cur_op, 2).o;
MVMString *name = MVM_cu_string(tc, cu, GET_UI32(cur_op, 4));
cur_op += 8;
MVM_6model_find_method(tc, obj, name, res, 1);
goto NEXT;
}
Thus, complex ops...

```
OP(findmeth): {
    /* Increment PC first, as we may make a method call. */
    MVMRegister *res = &GET_REG(cur_op, 0);
    MVMObject *obj = GET_REG(cur_op, 2).o;
    MVMString *name = MVM_cu_string(tc, cu, GET_UI32(cur_op, 4));
    cur_op += 8;
    MVM_6model_find_method(tc, obj, name, res, 1);
    goto NEXT;
}
```

Look up in a cache...and if it's not found, call the find_method method on the meta-object to find it...
Thus comes...

```c
OP(findmeth): {
    /* Increment PC first, as we may make a method call. */
    MVMRegister *res = &GET_REG(cur_op, 0);
    MVMObject *obj = GET_REG(cur_op, 2).o;
    MVMString *name = MVM_cu_string(tc, cu, GET_UI32(cur_op, 4));
    cur_op += 8;
    MVM_6model_find_method(tc, obj, name, res, 1);
    goto NEXT;
}
```

Look up in a cache...and if it's not found, call the find_method method on the meta-object to find it...

Wait, but then I need to find the find_method method...it'd better be in the cache...
Thus comes:

```
OP(findmeth):
 /* Increment PC first, as we may make a method call. */
MVMRegister* res = &GET_REG(cur_op, 0);
MVMObject* obj = GET_REG(cur_op, 2).o;
MVMString* name = MVM_cu_string(tc, cu, GET_UI32(cur_op, 4));
cur_op += 8;
MVM_6model_find_method(tc, obj, name, res, 1);
goto NEXT;
```

Wait, but then I need to find the find_method method...it'd

```
Oh, and the interpreter should not be recursively entered, so have to write the C code in
continuation passing style!
```

Look up in a cache...and if it's not found, call the find_method method on the meta-object to find it...
Here's another one

OP(if_o):
    GC_SYNC_POINT(tc);
    MVM_coerce_istrue(tc, GET_REG(cur_op, 0).o, NULL,
                       bytecode_start + GET_UI32(cur_op, 2),
                       cur_op + 6,
                       0);
    goto NEXT;

The thing that underlies if statements on objects today
Here's another one

Try to avoid making a method call to `Bool` when possible, because those were expensive.
void MVM_coerce_istrue(MVMThreadContext *tc, MVMObject *obj,
   MVMRegister *res_reg, MVMuint8 *true_addr, MVMuint8 *false_addr,
   MVMuint8 flip) {
MVMint64 result = 0;
if (!MVM_is_null(tc, obj)) {
   MVMBoolificationSpec *bs = obj->st->boolification_spec;
   switch (bs == NULL ? MVM_BOOL_MODE_NOT_TYPE_OBJECT : bs->mode) {
      case MVM_BOOL_MODE_UNBOX_INT:
         result = !IS_CONCRETE(obj) || REPR(obj)->box_funcs.get_int(tc,
                        STABLE(obj), obj, OBJECT_BODY(obj)) == 0 ? 0 : 1;
         break;
      case MVM_BOOL_MODE_UNBOX_NUM:
         result = !IS_CONCRETE(obj) || REPR(obj)->box_funcs.get_num(tc,
                        STABLE(obj), obj, OBJECT_BODY(obj)) == 0.0 ? 0 : 1;
         break;
...
void MVM_coerce_istrue(MVMThreadContext *tc, MVMObject *obj,
    MVMRegister *res_reg, MVMuint8 *true_addr, MVMuint8 *false_addr,
    MVMuint8 *flip) {
    MVMint64 result = 0;
    if (!MVM_is_null(tc, obj)) {
        MVMBoolificationSpec *bs = obj->st->boolification_spec;
        switch (bs == NULL ? MVM_BOOL_MODE_NOT_TYPE_OBJECT : bs->mode) {
            case MVM_BOOL_MODE_UNBOX_INT:
                result = !IS_CONCRETE(obj) || REPR(obj)->box_funcs.get_int(tc,
                    STABLE(obj), obj->OBJECT_BODY(obj)) == 0 ? 0 : 1;
                break;
            case MVM_BOOL_MODE_UNBOX_NUM:
                result = !IS_CONCRETE(obj) || REPR(obj)->box_funcs.get_num(tc,
                    STABLE(obj), obj->OBJECT_BODY(obj)) == 0.0 ? 0 : 1;
                break;
            ...
        }
    }
    ...

Another switch statement to decide what to do...but it's C so it's fast? 😊
With time, the runtime started to learn the tricks of the trade...
Type specialization
Record what types actually show up, produce optimized bytecode for those

Deoptimization
If the types are wrong, fall back to the original code

Inlining
Copy small routines into the caller, saving call costs

JIT compilation
Produce machine code, avoiding interpreter overhead
At first...

But now...

<table>
<thead>
<tr>
<th>At first...</th>
<th>But now...</th>
</tr>
</thead>
<tbody>
<tr>
<td>We only had a bytecode interpreter</td>
<td>We can JIT-compile hot bytecode into machine code</td>
</tr>
<tr>
<td>Lots of little method calls were prohibitively expensive</td>
<td>We can inline small method calls, so the calling cost is gone</td>
</tr>
<tr>
<td>Doing the hot-path decision making in C was a clear win</td>
<td>The C code is an opaque blob that we can't type specialize</td>
</tr>
</tbody>
</table>
Another issue: only the most common kinds of dispatch got special treatment in the VM
Hi! I'm a performance cliff!
Hi! I'm a performance cliff!

Multi-dispatch on nominal types

Method calls

Boolification
Hi! I'm a performance cliff!

- Multi-dispatch on nominal types
- Method calls
- Boolification
- Multi-dispatch with where clauses
- Qualified method calls
- callsame & co.
- .?method calls
In general:

If it's a dispatch, but the runtime (and especially the optimizer) can't reason about it as one, it'll probably be slow.
But why?
If we implement this:

```
my $result = $obj.?method($arg, $arg);
```

With a helper like this:

```
method dispatch:<.?>(Mu \SELF: Str() $name, |c) is raw { 
  nqp::can(SELF,$name) ??
  SELF."$name"(|c) !
  Nil
}
```
Mild polymorphism

my $result = $obj.?method($arg, $arg);

Becomes megamorphism

method dispatch:<.?>(Mu \SELF: Str() $name, |c) is raw {
    nqp::can($SELF,$name) ??
    $SELF."$name"(|c) !!
    Nil
}
Mild polymorphism

```perl
my $result = $obj.?method($arg, $arg);
```

One name, probably a handful of types

Becomes megamorphism

```perl
method dispatch:<.?>(Mu \SELF: Str() $name, |c) is raw {
    nqp::can(SELFL$name) ??
    SELF."$name"(|c) !!
    Nil
}
```

Many names, many types
Optimization relies heavily on turning potentially polymorphic programs into mostly monomorphic programs.
And to make matters even worse...

```r
method dispatch:<.?>(Mu \SELF: Str() $name, |c) is raw {
  nqp::can(SELF,$name) ??
  SELF."$name"(|c) !!
  Nil
}
```

Slurp here...

...flatten here
method dispatch:<.?>(Mu \SELF: Str() $name, |c) is raw {
    nqp::can(SEL, $name) ??
    SELF."$name"(|c) !!
    Nil
}

And to make matters even worse...

Callsite shapes are keys to specializations and caches, but are lost too

Slurp here...

...flatten here
So what can we do?
Teach the VM about more language features?
THE TYPE SPECIALIZER

BYTECODE, INLINE CACHES, ETC.

STUFF IN A C FUNCTION
ME

NOT HAVING TO HUNT GC BUGS

NEW GC BUGS
NEW LANGUAGE FEATURES

PROGRAMMABLE DISPATCH MECHANISM

MODIFYING THE VM
A new generalized approach to implementing the many different kinds of dispatch
One* new instruction

dispatch

* Actually, it comes in 5 forms by return type: void, native int, native num, native str, and object
Exposed like other ops

nqp::dispatch(...)
say(nqp::dispatch('boot-value', 42));  # 42
say(nqp::dispatch('boot-constant', 101));  # 101
say(nqp::dispatch('boot-value', 42));  # 42
say(nqp::dispatch('boot-constant', 101));  # 101
say(nqp::dispatch('boot-value', 42));  # 42
say(nqp::dispatch('boot-constant', 101));  # 101
sub spot-the-difference($x) {
    say(nqp::dispatch('boot-value', $x) ~ ' ' ~
        nqp::dispatch('boot-constant', $x));
}

spot-the-difference(1); # 1 1
spot-the-difference(2); # 2 1
spot-the-difference(3); # 3 1
sub spot-the-difference($x) {
    say(nqp::dispatch('boot-value', $x) ~ ' ' ~ 
         nqp::dispatch('boot-constant', $x));
}

spot-the-difference(1);    # 1 1
spot-the-difference(2);    # 2 1
spot-the-difference(3);    # 3 1

Some kind of cache?
sub spot-the-difference($x) {
    say(nqp::dispatch('boot-value', $x) ~ ' ' ~
         nqp::dispatch('boot-constant', $x));
}

spot-the-difference(1);     # 1 1
spot-the-difference(2);     # 2 1
spot-the-difference(3);

Let's look at the bytecode this compiles into!
sub spot-the-difference($x) {
    say(nqp::dispatch('boot-value', $x) ~ ' ' ~ nqp::dispatch('boot-constant', $x));
}
sub spot-the-difference($x) {
  say(nqp::dispatch('boot-value', $x) ~ ' ' ~
      nqp::dispatch('boot-constant', $x));
}

sub spot-the-difference($x) {
    say(nqp::dispatch('boot-value', $x) ~ ' ' ~ nqp::dispatch('boot-constant', $x));
}

00000  checkarity     1, 1
00001  param_rp_o     loc_0_obj, 0
00002  paramnamesused
    annotation: op.nqp:2
00003  getlex_no      loc_1_obj, '&say'
00004  dispatch_o     loc_2_obj, 'boot-value', Callsite_0, loc_0_obj
00005  decont          loc_2_obj, loc_2_obj
00006  dispatch_s     loc_3_str, 'nqp-stringify', Callsite_0, loc_2_obj
00007  const_s        loc_4_str, '
00008  concat_s       loc_4_str, loc_3_str, loc_4_str
00009  dispatch_o     loc_2_obj, 'boot-constant', Callsite_0, loc_0_obj
00010  dispatch_s     loc_3_str, 'nqp-stringify', Callsite_0, loc_2_obj
00011  concat_s       loc_3_str, loc_4_str, loc_3_str
00012  dispatch_o     loc_1_obj, 'lang-call', Callsite_1, loc_1_obj, loc_1_obj, loc_3_str
00013  return_o      loc_1_obj
sub spot-the-difference($x) {
    say(~nqp::dispatch('boot-value', $x) ~ ' ' ~ ~nqp::dispatch('boot-constant', $x));
}

checkarity 1, 1
param_rp_o loc_0_obj, 0
paramnamesused annotation: op.nqp:2
getlex_no loc_1_obj, '&say'
dispatch_o loc_2_obj, 'boot-value', Callsite_0, loc_0_obj
decont loc_2_obj, loc_2_obj
dispatch_s loc_3_str, 'nqp-stringify', Callsite_0, loc_2_obj
const_s loc_4_str, '
concat_s loc_4_str, loc_3_str, loc_4_str
dispatch_o loc_2_obj, 'boot-constant', Callsite_0, loc_0_obj
dispatch_s loc_3_str, 'nqp-stringify', Callsite_0, loc_2_obj
concat_s loc_3_str, loc_4_str, loc_3_str
dispatch_o loc_1_obj, 'lang-call', Callsite_1, loc_1_obj, loc_1_obj, loc_3_str
return_o loc_1_obj
checkarity 1, 1
param_rp_o loc_0_obj, 0
paramnamesused
  annotation: op.nqp:2
getlex_no loc_1_obj, '&say'
dispatch_o loc_2_obj, 'boot-value', Callsite_0, loc_0_obj
decont loc_2_obj, loc_2_obj
dispatch_s loc_3_str, 'nqp-stringify', Callsite_0, loc_2_obj
const_s loc_4_str, '
concat_s loc_4_str, loc_3_str, loc_4_str
dispatch_o loc_2_obj, 'boot-constant', Callsite_0, loc_0_obj
dispatch_s loc_3_str, 'nqp-stringify', Callsite_0, loc_2_obj
concat_s loc_3_str, loc_4_str, loc_3_str
dispatch_o loc_1_obj, 'lang-call', Callsite_1, loc_1_obj, loc_3_str
return_o loc_1_obj
On the first call, allocate 1 pointer of storage for each dispatch op...
...and initialize all of them to a pointer to a singleton "unlinked" state

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Location</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkarity</td>
<td>0x00000c</td>
<td>1, 1</td>
</tr>
<tr>
<td>param_rp_o</td>
<td>0x000011</td>
<td>loc_0_obj, 0</td>
</tr>
<tr>
<td>paramnamesused</td>
<td>annotation: op.nqp:2</td>
<td></td>
</tr>
<tr>
<td>getlex_no</td>
<td>loc_1_obj, '&amp;say'</td>
<td></td>
</tr>
<tr>
<td>dispatch_o</td>
<td>loc_2_obj, 'boot-value', Callsite_0, loc_0_obj</td>
<td></td>
</tr>
<tr>
<td>decont</td>
<td>loc_2_obj, loc_2_obj</td>
<td></td>
</tr>
<tr>
<td>dispatch_s</td>
<td>loc_3_str, 'nqp-stringify', Callsite_0, loc_2_obj</td>
<td></td>
</tr>
<tr>
<td>const_s</td>
<td>loc_4_str, ''</td>
<td></td>
</tr>
<tr>
<td>concat_s</td>
<td>loc_4_str, loc_3_str, loc_4_str</td>
<td></td>
</tr>
<tr>
<td>dispatch_o</td>
<td>loc_2_obj, 'boot-constant', Callsite_0, loc_0_obj</td>
<td></td>
</tr>
<tr>
<td>dispatch_s</td>
<td>loc_3_str, 'nqp-stringify', Callsite_0, loc_2_obj</td>
<td></td>
</tr>
<tr>
<td>concat_s</td>
<td>loc_3_str, loc_4_str, loc_3_str</td>
<td></td>
</tr>
<tr>
<td>dispatch_o</td>
<td>loc_1_obj, 'lang-call', Callsite_1, loc_1_obj, loc_3_str</td>
<td></td>
</tr>
<tr>
<td>return_o</td>
<td>loc_1_obj</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unlinked</th>
<th>Unlinked</th>
<th>Unlinked</th>
<th>Unlinked</th>
<th>Unlinked</th>
<th>Unlinked</th>
</tr>
</thead>
</table>
What does dispatch do?

```
OP(dispatch_o): {
    MVMDispInlineCacheEntry **ice_ptr = MVM_disp_inline_cache_get(
        cur_op, bytecode_start, tc->cur_frame);
    MVMDispInlineCacheEntry *ice = *ice_ptr;
    MVMString *id = MVM_cu_string(tc, cu, GET_UI32(cur_op, 2));
    MVMCallsite *callsite = cu->body.callsites[GET_UI16(cur_op, 6)];
    MVMuint16 *args = (MVMuint16 *)(cur_op + 8);
    MVMuint32 bytecode_offset = cur_op - *tc->interp_bytecode_start - 2;
    tc->cur_frame->return_value = &GET_REG(cur_op, 0);
    tc->cur_frame->return_type = MVM_RETURN_OBJ;
    cur_op += 8 + 2 * callsite->flag_count;
    tc->cur_frame->return_address = cur_op;
    ice->run_dispatch(tc, ice_ptr, ice, id, callsite, args,
        tc->cur_frame->work, tc->cur_frame->static_info,
        bytecode_offset);
    goto NEXT;
}
```
What does dispatch do?

OP(dispatch_o): {
    MVMDispInlineCacheEntry **ice_ptr = MVM_disp_inline_cache_get(
        cur_op, bytecode_start, tc->cur_frame);
    MVMDispInlineCacheEntry *ice = *ice_ptr;
    MVMString *id = MVM_cu_string(tc, cu, GET_UI32(cur_op, 2));
    MVMCallsite *callsite = cu->body.callsites[GET.UI16(cur_op, 6)];
    MVMuint16 *args = (MVMuint16 *)(cur_op + 8);
    MVMuint32 bytecode_offset = cur_op - *tc->interp_bytecode_start - 2;
    tc->cur_frame->return_value = &GET_REG(cur_op, 0);
    tc->cur_frame->return_type = MVM_RETURN_OBJ;
    cur_op += 8 + 2 * callsite->flag_count;
    tc->cur_frame->return_address = cur_op;
    ice->run_dispatch(tc, ice_ptr, ice, id, callsite, args,
        tc->cur_frame->work, tc->cur_frame->static_info,
        bytecode_offset);
    goto NEXT;
}

Find the address of the current state
What does dispatch do?

OP(dispatch_o): {
    MVMDispInlineCacheEntry **ice_ptr = MVM_disp_inline_cache_get(
        cur_op, bytecode_start, tc->cur_frame);
    MVMDispInlineCacheEntry *ice = *ice_ptr;
    MVMString *id = MVM_cu_string(tc, cu, GET_UI32(cur_op, 2));
    MVMCallsite *callsite = cu->body.callsites[GET_UI16(cur_op, 6)];
    MVMuint16 *args = (MVMuint16 *)(cur_op + 8);
    MVMuint32 bytecode_offset = cur_op - *tc->interp_bytecode_start - 2;
    tc->cur_frame->return_value = &GET_REG(cur_op, 0);
    tc->cur_frame->return_type = MVM_RETURN_OBJ;
    cur_op += 8 + 2 * callsite->flag_count;
    tc->cur_frame->return_address = cur_op;
    ice->run_dispatch(tc, ice_ptr, ice, id, callsite, args,
        tc->cur_frame->work, tc->cur_frame->static_info,
        bytecode_offset);
    goto NEXT;
}

Dereference it to get the current state
What does dispatch do?

OP(dispatch_o): {
    MVMDispInlineCacheEntry **ice_ptr = MVM_disp_inline_cache_get(cur_op, bytecode_start, tc->cur_frame);
    MVMDispInlineCacheEntry *ice = *ice_ptr;
    MVMString *id = MVM_cu_string(tc, cu, GET_UI32(cur_op, 2));
    MVMCallsite *callsite = cu->body.callsites[GET_UI16(cur_op, 6)];
    MVMuint16 *args = (MVMuint16 *)(cur_op + 8);
    MVMuint32 bytecode_offset = cur_op - *tc->interp_bytecode_start - 2;
    tc->cur_frame->return_value = &GET_REG(cur_op, 0);
    tc->cur_frame->return_type = MVM_RETURN_OBJ;
    cur_op += 8 + 2 * callsite->flag_count;
    tc->cur_frame->return_address = cur_op;
    ice->run_dispatch(tc, ice_ptr, ice, id, callsite, args,
        tc->cur_frame->work, tc->cur_frame->static_info,
        bytecode_offset);
    goto NEXT;
}

Look up the dispatcher ID and callsite shape
What does dispatch do?

OP(dispatch_o): {
    MVMDispInlineCacheEntry **ice_ptr = MVM_disp_inline_cache_get(
        cur_op, bytecode_start, tc->cur_frame);
    MVMDispInlineCacheEntry *ice = *ice_ptr;
    MVMString *id = MVM_cu_string(tc, cu, GET_UI32(cur_op, 2));
    MVMCallsite *callsite = cu->body.callsites[GET_UI16(cur_op, 6)];
    MVMuint16 *args = (MVMuint16 *)(cur_op + 8);
    MVMuint32 bytecode_offset = cur_op - *tc->interp_bytecode_start - 2;
    tc->cur_frame->return_value = &GET_REG(cur_op, 0);
    tc->cur_frame->return_type = MVM_RETURN_OBJ;
    cur_op += 8 + 2 * callsite->flag_count;
    tc->cur_frame->return_address = cur_op;
    ice->run_dispatch(tc, ice_ptr, ice, id, callsite, args,
        tc->cur_frame->work, tc->cur_frame->static_info,
        bytecode_offset);
    goto NEXT;
}
What does dispatch do?

OP(dispatch_o): {
    MVMDispInlineCacheEntry **ice_ptr = MVM_disp_inline_cache_get(
        cur_op, bytecode_start, tc->cur_frame);
    MVMDispInlineCacheEntry *ice = *ice_ptr;
    MVMString *id = MVM_cu_string(tc, cu, GET_UI32(cur_op, 2));
    MVMCallsite *callsite = cu->body.callsites[GET_UI16(cur_op, 6)];
    VMUint16 *args = (VMUint16 *)(cur_op + 8);
    VMUint32 bytecode_offset = cur_op - *tc->interp_bytecode_start - 2;
    tc->cur_frame->return_value = &GET_REG(cur_op, 0);
    tc->cur_frame->return_type = MVM_RETURN_OBJ;
    cur_op += 8 + 2 * callsite->flag_count;
    tc->cur_frame->return_address = cur_op;
    ice->run_dispatch(tc, ice_ptr, ice, id, callsite, args,
        tc->cur_frame->work, tc->cur_frame->static_info,
        bytecode_offset);
    goto NEXT;
}
What does dispatch do?

OP(dispatch_o): {
    MVMDispInlineCacheEntry **ice_ptr = MVM_disp_inline_cache_get(
        cur_op, bytecode_start, tc->cur_frame);
    MVMDispInlineCacheEntry *ice = *ice_ptr;
    MVMString *id = MVM_cu_string(tc, cu, GET_UI32(cur_op, 2));
    MVMCallsite *callsite = cu->body.callsites[GET_UI16(cur_op, 6)];
    MVMuint16 *args = (MVMuint16 *)(cur_op + 8);
    MVMuint32 bytecode_offset = cur_op - *tc->interp_bytecode_start - 2;
    tc->cur_frame->return_value = &GET_REG(cur_op, 0);
    tc->cur_frame->return_type = MVM_RETURN_OBJ;
    cur_op += 8 + 2 * callsite->flag_count;
    tc->cur_frame->return_address = cur_op;
    ice->run_dispatch(tc, ice_ptr, ice, id, callsite, args,
        tc->cur_frame->work, tc->cur_frame->static_info,
        bytecode_offset);
    goto NEXT;
}
In the unlinked state, we record a dispatch program
sub spot-the-difference($x) {
    say(nqp::dispatch('boot-value', $x) ~ ' ' ~ 
         nqp::dispatch('boot-constant', $x));
}

Dispatch program 0x55f437d2b2b0 (1 temporaries)
   at op.nqp:2 (<ephemeral file>:spot-the-difference)
Ops:
   Load argument 0 into temporary 0
   Set result object value from temporary 0
sub spot-the-difference($x) {
    say(nqp::dispatch('boot-value', $x) ~ ' ' ~ nqp::dispatch('boot-constant', $x));
}

Dispatch program 0x55f437d2baa0 (1 temporaries)
at op.nqp:3 (<ephemeral file>:spot-the-difference)

Ops:
  Load collectable constant at index 0 into temporary 0
  Set result object value from temporary 0
boot-constant and boot-value are two of the dispatch terminals - things that produce a final outcome
# Run bytecode with some arguments (works only in NQP since this is a raw bytecode handle, not a Code object as in Raku)

my $code := -> $x, $y { say($x + $y) };

nqp::dispatch('boot-code', $code, 40, 2); # 42
# Calls VM-provided functionality

my @arr := [1,2,3];
say(nqp::dispatch('boot-syscall', 'elems', # 3 @arr));
That's all we need.

(Because everything else can be exposed in terms of boot-syscall.)
Userspace dispatchers

Additional dispatchers can be registered using a syscall

nqp::dispatch('boot-syscall', 'dispatcher-register', 'identity',
    # Invoked with the argument capture
    -> $capture {
        ...
    });
Delegation

User-defined dispatchers always delegate to some other dispatcher

```
nqp::dispatch('boot-syscall', 'dispatcher-register', 'identity',
  # Invoked with the argument capture
  -> $capture {
    # Must delegate, ultimately to a terminal
    nqp::dispatch('boot-syscall', 'dispatcher-delegate',
      'boot-value', $capture);
  });
```
They show up in the bytecode just like any built-in dispatcher

```perl
nqp::dispatch('boot-syscall', 'dispatcher-register', 'identity',
    # Invoked with the argument capture
    -> $capture {
        # Must delegate, ultimately to a terminal
        nqp::dispatch('boot-syscall', 'dispatcher-delegate',
            'boot-value', $capture);
    });

say(nqp::dispatch('identity', 'function')); # function
```
Capture transforms

Insert argument
Drop argument

```perl
nqp::dispatch('boot-syscall', 'dispatcher-register', 'drop-first',
-> $capture {
    my $shorter := nqp::dispatch('boot-syscall',
        'dispatcher-drop-arg', $capture, 1);
    nqp::dispatch('boot-syscall', 'dispatcher-delegate',
        'boot-code', $shorter);
});

my $code := -> $a { $a }
say(nqp::dispatch('drop-first', $code, 1, 2));  # 2
```
What about this one?

```perl
nqp::dispatch('boot-syscall', 'dispatcher-register', 'type-name',
   -> $capture {
      # Get the type name.
      my $arg := nqp::captureposarg($capture, 0);
      my str $name := $arg.HOW.name($arg);
      # Evaluate to the type name.
      my $outcome := nqp::dispatch('boot-syscall',
         'dispatcher-insert-arg-literal-str',
         $capture, 0, $name);
      nqp::dispatch('boot-syscall', 'dispatcher-delegate',
         'boot-constant', $outcome);
   });
```
Oops!

class Badger {}
class Mushroom {}
sub try-it($obj) {
    nqp::dispatch('type-name', $obj)
}
say(try-it(Badger));  # Badger
say(try-it(Badger));  # Badger
say(try-it(Badger));  # Badger
say(try-it(Badger));  # Badger
say(try-it(Badger));  # Badger
say(try-it(Mushroom)); # Badger
say(try-it(Mushroom)); # Badger
Need to add a guard

```perl
nqp::dispatch('boot-syscall', 'dispatcher-register', 'type-name', -> $capture {
    # Get the type name.
    my $arg := nqp::captureposarg($capture, 0);
    my str $name := $arg.HOW.name($arg);
    # Guard by type.
    my $track-arg := nqp::dispatch('boot-syscall', 'dispatcher-track-arg', $capture, 0);
    nqp::dispatch('boot-syscall', 'dispatcher-guard-type', $track-arg);
    # Evaluate to the type name.
    my $outcome := nqp::dispatch('boot-syscall', 'dispatcher-insert-arg-literal-str', $capture, 0, $name);
    nqp::dispatch('boot-syscall', 'dispatcher-delegate', 'boot-constant', $outcome);
});
```
Better!

class Badger {}
class Mushroom {}
sub try-it($obj) {
    nqp::dispatch('type-name', $obj)
}
say(try-it(Badger));  # Badger
say(try-it(Badger));  # Badger
say(try-it(Badger));  # Badger
say(try-it(Badger));  # Badger
say(try-it(Badger));  # Badger
say(try-it(Badger));  # Badger
say(try-it(Mushroom)); # Mushroom
say(try-it(Mushroom)); # Mushroom
Dispatch programs erase all delegation, all intermediate captures, and duplicate guards!

Dispatch program 0x559750f6d000 (1 temporaries)
at op.nqp:21 (<ephemeral file>:try-it)

Ops:
Guard arg 0 (type=Badger)
Load collectable constant at index 1 into temporary 0
Set result string value from temporary 0
Callsite transitions

Unlinked
Callsite transitions

Unlinked

Monomorphic

Guard arg 0 (type=Badger)
Load collectable constant at index 1 into temporary 0
Set result string value from temporary 0
Callsite transitions

- **Unlinked**

- **Monomorphic**
  - Guard arg 0 (type=Badger)
  - Load collectable constant at index 1 into temporary 0
  - Set result string value from temporary 0

- **Polymorphic**
  - Guard arg 0 (type=Badger)
  - Load collectable constant at index 1 into temporary 0
  - Set result string value from temporary 0
  - Guard arg 0 (type=Mushroom)
  - Load collectable constant at index 1 into temporary 0
  - Set result string value from temporary 0
A real example

```
nqp::dispatch('boot-syscall', 'dispatcher-register', 'nqp-meth-call', -> $capture {
    # Try to find the method; complain if there's none found.
    my $obj := nqp::captureposarg($capture, 0);
    my str $name := nqp::captureposarg_s($capture, 1);
    my $meth := $obj.HOW.find_method($obj, $name);
    if nqp::isconcrete($meth) {
        # Establish a guard on the invocant type and method name (however the name
        # may well be a literal, in which case this is free).
        nqp::dispatch('boot-syscall', 'dispatcher-guard-type',
            nqp::dispatch('boot-syscall', 'dispatcher-track-arg', $capture, 0));
        nqp::dispatch('boot-syscall', 'dispatcher-guard-literal',
            nqp::dispatch('boot-syscall', 'dispatcher-track-arg', $capture, 1));

        # Drop the first two arguments, which are the decontainerized invocant
        # and the method name. Then insert the resolved method and delegate to
        # lang-call to invoke it (we may have other languages mixing into NQP
        # types and adding their methods).
        my $args := nqp::dispatch('boot-syscall', 'dispatcher-drop-arg',
            nqp::dispatch('boot-syscall', 'dispatcher-drop-arg', $capture, 0),
            0);
        my $delegate := nqp::dispatch('boot-syscall', 'dispatcher-insert-arg-literal-obj',
            $args, 0, $meth);
        nqp::dispatch('boot-syscall', 'dispatcher-delegate', 'lang-call', $delegate);
    } else {
        nqp::dispatch('boot-syscall', 'dispatcher-delegate', 'lang-meth-not-found', $capture);
    }
});
```
A real example

```perl
nqp::dispatch('boot-syscall', 'dispatcher-register', 'nqp-meth-call', \-> $capture {
  # Try to find the method.
  my $obj := nqp::captureposarg($capture, 0);
  my str $name := nqp::captureposarg_s($capture, 1);
  my $meth := $obj.HOW.find_method($obj, $name);
  if nqp::isconcrete($meth) {
    # Establish a guard on the invocant type and method name (however the name
    # may well be a literal, in which case this is free).
    nqp::dispatch('boot-syscall', 'dispatcher-guard-type',
      nqp::dispatch('boot-syscall', 'dispatcher-track-arg', $capture, 0));
    nqp::dispatch('boot-syscall', 'dispatcher-guard-literal',
      nqp::dispatch('boot-syscall', 'dispatcher-track-arg', $capture, 1));
    # Drop the first two arguments, which are the decontainerized invocant
    # and the method name. Then insert the resolved method and delegate to
    # lang-call to invoke it (we may have other languages mixing into NQP
    # types and adding their methods).
    my $args := nqp::dispatch('boot-syscall', 'dispatcher-drop-arg',
      nqp::dispatch('boot-syscall', 'dispatcher-drop-arg', $capture, 0),
      0);
    my $delegate := nqp::dispatch('boot-syscall', 'dispatcher-insert-arg-literal-obj',
      $args, 0, $meth);
    nqp::dispatch('boot-syscall', 'dispatcher-delegate', 'lang-call', $delegate);
  }
  else {
    nqp::dispatch('boot-syscall', 'dispatcher-delegate', 'lang-meth-not-found', $capture);
  }
});
```
A real example

```perl
nqp::dispatch('boot-syscall', 'dispatcher-register', 'nqp-meth-call', -> $capture {
    # Try to find the method.
    my $obj := nqp::captureposarg($capture, 0);
    my str $name := nqp::captureposarg_s($capture, 1);
    my $meth := $obj.HOW.find_method($obj, $name);
    if nqp::isconcrete($meth) {
        # Establish a guard on the invocant type and method name (however the name
        # may well be a literal, in which case this is free).
        nqp::dispatch('boot-syscall', 'dispatcher-guard-type',
            nqp::dispatch('boot-syscall', 'dispatcher-track-arg', $capture, 0));
        nqp::dispatch('boot-syscall', 'dispatcher-guard-literal',
            nqp::dispatch('boot-syscall', 'dispatcher-track-arg', $capture, 1));
        # Drop the first two arguments, which are the decontainerized invocant
        # and the method name. Then insert the resolved method and delegate to
        # lang-call to invoke it (we may have other languages mixing into NQP
        # types and adding their methods).
        my $args := nqp::dispatch('boot-syscall', 'dispatcher-drop-arg',
            nqp::dispatch('boot-syscall', 'dispatcher-drop-arg', $capture, 0),
            0);
        my $delegate := nqp::dispatch('boot-syscall', 'dispatcher-insert-arg-literal-obj',
            $args, 0, $meth);
        nqp::dispatch('boot-syscall', 'dispatcher-delegate', 'lang-call', $delegate);
    } else {
        nqp::dispatch('boot-syscall', 'dispatcher-delegate', 'lang-meth-not-found', $capture);
    }
});
```
A real example

nqp::dispatch('boot-syscall', 'dispatcher-register', 'nqp-meth-call', -> $capture {
  # Try to find the method.
  my $obj := nqp::captureposarg($capture, 0);
  my str $name := nqp::captureposarg_s($capture, 1);
  my $meth := $obj.HOW.find_method($obj, $name);
  if nqp::isconcrete($meth) {
    # Establish a guard on the invocant type and method name (however the name
    # may well be a literal, in which case this is free).
    nqp::dispatch('boot-syscall', 'dispatcher-guard-type',
      nqp::dispatch('boot-syscall', 'dispatcher-track-arg', $capture, 0));
    nqp::dispatch('boot-syscall', 'dispatcher-guard-literal',
      nqp::dispatch('boot-syscall', 'dispatcher-track-arg', $capture, 1));
  }
  else {
    nqp::dispatch('boot-syscall', 'dispatcher-delegate',
      lang-meth-not-found, $capture);
  }
});

# Establish a guard on the invocant type and method name (however the name
# may well be a literal, in which case this is free).
nqp::dispatch('boot-syscall', 'dispatcher-guard-type',
  nqp::dispatch('boot-syscall', 'dispatcher-track-arg', $capture, 0));
nqp::dispatch('boot-syscall', 'dispatcher-guard-literal',
  nqp::dispatch('boot-syscall', 'dispatcher-track-arg', $capture, 1));

# Drop the first two arguments, which are the decontainerized invocant

# Establish a guard on the invocant type and method name
# (however the name may well be a literal, in which case this
# is free).

A real example

my $args := nqp::dispatch('boot-syscall', 'dispatcher-drop-arg',
    nqp::dispatch('boot-syscall', 'dispatcher-drop-arg',
        $capture, 0),
    0);
my $delegate := nqp::dispatch('boot-syscall',
    'dispatcher-insert-arg-literal-obj',
    $args, 0, $meth);
nqp::dispatch('boot-syscall', 'dispatcher-delegate', 'lang-call',
    $delegate);

# and the method is resolved.
# types and adding their methods).
my $args := nqp::dispatch('boot-syscall', 'dispatcher-drop-arg',
    nqp::dispatch('boot-syscall', 'dispatcher-drop-arg', $capture, 0),
    0);
my $delegate := nqp::dispatch('boot-syscall', 'dispatcher-insert-arg-literal-obj',
    $args, 0, $meth);
nqp::dispatch('boot-syscall', 'dispatcher-delegate', 'lang-call', $delegate);
}
else {
    nqp::dispatch('boot-syscall', 'dispatcher-delegate', 'lang-meth-not-found', $capture);
}};
Resumption

Dispatchers also support saving state - in the best case at zero runtime cost - in case a dispatch is resumed.

Used for callsame, nextwith, etc.

Also used for multis with where clauses, unpacking, etc.
And then...

Dispatch programs in hot code are translated into ops by the optimizer

Then we optimize: inlining, larger scale guard elimination, etc.

JIT compilation of what remains, removing interpretation overhead
The current status
in terms of completion and performance
Currently passing 99.6% of specification tests
Issues remain with < 40 ecosystem modules
"Make it work, then make it fast"
Still need to re-enable some really important optimizations

Without them, we will obviously do poorly at basic cases of method dispatch and multiple dispatch

They've had so much attention in the current implementation
But we can find some things to measure...
Baseline performance
(master vs. new-disp, optimizer disabled)
Baseline performance
(master vs. new-disp, optimizer disabled)

class C {
    method m() { }
}
for 1..10_000_000 {
    C.m
}
Baseline performance (master vs. new-disp, optimizer disabled)

```
my $x = 0;
for 1..10_000_000 {
    $x = $x + $_;
}
```
Baseline performance
(master vs. new-disp, optimizer disabled)

Competitive, and there's still some obvious unimplemented improvements to do

![Bar chart showing performance comparison between master and new-disp for method calls and multi call.](chart.png)
Baseline performance
(master vs. new-disp, optimizer disabled)

Surprisingly good given a general mechanism has replaced a special purpose multi cache
What about the features that have especially weak performance on master?

Is the design improvement on new-disp enough to come out ahead, even with the optimizer integration incomplete?
Currently slow dispatches
(master vs. new-disp)
Currently slow dispatches
(master vs. new-disp)

multi fac($n where $n <= 1) { 1 }
multi fac($n) { $n * fac($n - 1) }
for ^100_000 {
  fac(10)}
Currently slow dispatches
(master vs. new-disp)

```
multi fallback(Any $x) { "a$x" }
multi fallback(Numeric $x) { "n" ~ callsame }
multi fallback(Real $x) { "r" ~ callsame }
multi fallback(Int $x) { "i" ~ callsame }
for ^1_000_000 {
    fallback(4+2i);
    fallback(4.2);
    fallback(42);
}
```
Currently slow dispatches
(master vs. new-disp)

class C {
   method CALL-ME($maybe) {}
}
my $c = C.new;
for 1..10_000_000 {
   $c(42);
}
Currently slow dispatches
(master vs. new-disp)

 Wins these handily, despite not yet being fully integrated back into the optimizer

<table>
<thead>
<tr>
<th>Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi with where</td>
</tr>
<tr>
<td>callsame multis</td>
</tr>
<tr>
<td>CALL-ME</td>
</tr>
</tbody>
</table>

- master
- new-disp
Future opportunities for further improvements
First, get it good enough to merge

- Final few spectests fixed
- Minimize ecosystem regressions
- Fully re-integrated into optimizer
- Minimize significant performance regressions
Optimization of dispatch resumptions

Inline small targets of a callsame

Make where clauses much more competitive with conditionals
Faster native calls

We could treat them as another kind of dispatch terminal

Argument marshalling could be largely done in dispatch programs

Potential for vast improvement on today, especially when JIT is involved
And much more...

Faster calls on role puns

Faster delegation (handles)

Faster FALLBACK method handling

A safe "call if it binds" mechanism (faster Cro request routing)
Thank you!

@ jonathan@edument.cz
W jnthn.net
Twitter jnthnwrthngtn
GitHub jnthn